Comp 466 – Assignment #1

Moshiur Howlader

November 13, 2018

Homepage

The homepage was designed to contain the launching page to allow directory access to all the parts of the web applications implemented for this assignment. A clean, simplistic design was used for the overall look of the web application, with a navbar at the top for easy access to each part of the assignment. Also, a footer to cover the bottom of the webpage was implemented. Majority of the CSS design for the Web Application was implemented in this stage of the assignment. A small div container with a green border at the top left part of the homepage contains the cover page details for this assignment. An expandable/collapsible accordion containing summary outlining what each part of the assignment did was implemented. As well, two <a> tag buttons were included at the end of the webpage containing access to the assignment#1 instructions. Extra details were added to the homepage to add some contents so that the web page was not so empty.

Note, the documentation for this assignment is placed in the "documentation" tab of this website. Please navigate there to view it.

Part 1:

Hyperlink: `http://localhost/TMA_1/part1/part1.xsl`

Part 1 required an implementation of an XML file containing my resume. Three general sections were required, which were my general information, my educational background, and my work experience. As well, a schema, and the XSLT for the XML resume was required to be implemented. I interpreted this portion of the assignment as implementing the XML, the schema, and the XSLT into my general web application page tab of Part 1. In other words, I was to integrate my XML resume nicely with the overall look of this web application. To do so, an appropriate XSLT file or .xsl file was needed to be implemented (note XSLT is a subset of XSL file, so defining either extension did not change anything). As well, a schema was asked to be written for this XML.

To design the XML file, four sections were realized, a <Highlight>, <Education>, <Skills>, and <Work Experience>. As the tags suggests, appropriate highlights, education, skills, and work experience were included into a modularized section. The entire XML file was wrapped in <resume> tag. The schema that corresponds to this XML file was simply implemented into the above mentioned tags as <xs:element> with the corresponding tag names. Since resumes do not contain any data types other than strings, only string element types were defined in my schema. Finally, the implementation of the styling for the XML file was made in the part1.xsl file. The main challenge was to replicate the look and feel of the CSS counterpart to the web application while including the XML equivalent resume into this webpage, so some margin spacing, and positional stylings were needed to be adjusted; especially for the footer. Once the styling in the xsl file was complete, simply the retrieval of the XML data using xsl command of <xsl:value-of select=""> was used to retrieve the appropriate data.

To view this application, simply click the “Part 1” tab in the Navbar of the Web application to navigate to the hyperlink.

Part 2:

Hyperlink: `http://localhost/TMA_1/part2/part2.htm`

A learning web application with a built-in customizable client-side quiz system was requested to be implemented. Various features were requested: a welcome page with a banner, tutorial for the first 3 units of this course, default quiz for the 3 units, auto-grader for the quizzes with feedback, editable quiz module, use of all the major web technologies covered so far in the course, teach enough breadth in content, consistent external styling, and appropriate use of HTML5 tables. Only interpretation I used to implement this part was the quiz making system is client-side, so no communication with the server for generating or updating the XML file on server-side. This was clarified by Dr. Wang.

To implement the first requirement, a basic accordion was used as a welcome page to introduce the user to what this part of the webpage did. It is coloured in red to make it stand out. Once you hover over, it turns purple, and has a “Got It” button to close the accordion. The learning module was named “AvenueToQuiz” and the module contains a nice red banner to show this off.

The second requirement was implemented by having tabs for each unit. The user can navigate only one unit at a time to help not make the web application busy. The code used in the tutorial portion of the web app was styled in a standard manner to make code easy to view (using the code tag). HTML5 tables were used to list tabular data such as list of HTML elements. Relevant links were included, as well as a Youtube Video containing a summary of HTML elements. For unit 3 of the tutorial, a customized zooming functioning to view the documents were implemented to allow an easier viewing experience for the user. Otherwise, this requirement was straightforward, albeit time-consuming.

The third requirement was implemented by simply having the “quiz” tab, and adding the <a> tag to navigate to the “quiz” tab.

The fourth requirement was implemented by first having an onclick event which loads the XML file containing the standard, default test bank question, and loading it into the XMLHttpRequest object, which in our case was called xhttp. The xhttp object was used to parse out the relevant quiz questions contained in the XML file quizDatabase.xml based on the tag name “unit” and “quizName” attributes (unit1, unit 2, unit3) inside the loadQuiz#() function. Inside the loadQuiz() functions, a random number generator was used to create a random quiz from the given unit from a possible 13 questions. Appropriate array data structures were made, used, and reset to track each user quiz instance. Significant DOM usage was required to implement the quiz functionality. At the end of loadQuiz(), another function, displayQuizResult() was called to display the user’s attempt at the randomly generated quiz. The feedback displayed in a bar chart the percentage of correct question attempt (green) vs percentage of incorrect question attempt (red). This bar chart was implemented using a mix of DOM and CSS properties. Some basic logic was also implemented to give custom response based on user’s performance, and feedback containing all the questions attempted with the correct response for each of the quiz question is displayed.

The fifth requirement, the hardest requirement, was implemented by having the fifth tab ("Edit Quiz") in the tab menu of the "AvenueToQuiz" system. Here we have 3 buttons: "Edit an existing Quiz", "Create a new Quiz", and "Delete an existing Quiz". "Create a new Quiz" allows users to create their own new quiz with whatever name they choose. After the new blank quiz is created, it can be edited with "Edit an existing Quiz" to modify the blank quiz so that it contains a minimum of 7 questions. After populating the quiz content, you can navigate to the "Try Quiz" tab and select the quiz to attempt it. And simply, if you wish to delete your newly made quizzes, go to "Delete an existing Quiz" to select and delete that quiz. One large issue I had was every time the "Change" button was pressed during an edit of a quiz, when attempting the newly made quiz, every press of the "Change" button generated that many identical random quizzes in the single quiz attempt for the user. To eliminate this problem, I simply added an if statement to make sure that only to add the elements through DOM only if the question container and submit button count was 0. That was done so that the recursive calls of the event listener `implementQuizSys[quizContextIndex].addEventListener("click", function(event) {…}` did not flood the quiz attempt with multiple identical quizzes from the multiple changes/edits the user made on that one custom made quiz. Another major roadblock was the figuring out of the use of closures. Every time a new quiz was made, its own appropriate instance value of i was required to be bounded, otherwise, all the user made quizzes would end up sharing the i value. This i value without the use of closures is the total length of the quizList being assigned to every quiz in that quizList. Hence by using closures, the appropriate index value (0 to quizList.length-1) for each of the quizzes made by that user were assigned correctly. Also, closures were much needed to dynamically generate new quizzes. Other than the above two issues, standard debugging of data structures, and DOM were needed to be tracked carefully through use of Google Chrome's console to trace the flow of the part2.js

The remaining requirements were met, to the best of my ability.

To use this web app, for tutorial navigate to Unit 1, 2, or 3 tabs to learn the contents. Go to Try Quiz or Edit Quiz to attempt or create a quiz to test your understanding. Please refer to the fifth requirement on how to use the Edit Quiz portion of this assignment.

Part 3:

Hyperlink: `http://localhost/TMA_1/part3/part3.htm`

Please note that I have obtained the images from https://www.pexels.com/, and I only took pictures that were completely copyright free (free to use for any personal and commercial projects and also I have included the sources). I do not own a good camera, my apologies!

For the first requirement, a canvas of 640px by 480px was used for the slideshow.

For the second requirement, each image was given an appropriate caption. To draw the caption on the canvas, a custom function, `writeCaption()` was implemented which calculated the pixel position of the center bottom of the image, and drew the caption on a black background, in white font, symmetrically about the y-axis.

For the third requirement, a button was placed below the slideshow image to start or pause the slideshow. This button was implemented by attaching onclick event to toggle the `var = paused` flag. This var (`paused`) is used to control in the code how the slideshow should work.

For the fourth requirement, the rightmost button allows the user to toggle between random/shuffle and sequential mode. The button for this is assigned an onclick event to toggle the `var = shuffle_on` flag.

For the fifth requirement, there are two buttons (Prev and Next) placed besides the start/pause button that allows user to rewind or fast-forward through images only when the show is in sequential mode. This logic was implemented in code by checking before incrementing the image index of the slideshow, whether the `shuffle_on` mode was false. Note the entire slideshow transition logic was implemented in the `mySlideshowCallback()` function, which was set to an interval of 2.5 seconds.

For the sixth requirement, once the user selected the dropdown menu containing the transition effect options, the appropriate `transition_value` was checked within the event handler slide_image.onload = function(){…} to change how the canvas drew the images everytime the slide_image changed. Note, `slide_image.src = images.img[index]`, where `index` is the image index in the JSON array `images.img[ ]`. When the `transition_value` was 0, no effects to the transitions were applied; simply draw a new image every 2.5 seconds. When the `transition_value` was 1, a fading effect was applied. This effect was implemented by having a `reveal_timer` which was set to an interval of 0.1 seconds. And every 0.1 seconds, until 10 iterations of 0.1 seconds occurred, a `fadeInImage()` was called. `fadeInImage()` was used to gradually change the globalAlpha value which affected the transparency of the image being drawn in this 1 second transition interval. This smooth change in globalAlpha gives a smooth transition effect. Hence, every 1 seconds out of the 2.5 second slideshow Image change interval is spent doing a fading transition. This 1 second transition logic is reapplied for the other `transition_value` 2 and 3. `transition_value` 2 does a transition effect where the upcoming image slides in from the right while the old images stays put in the background. 3 does a transition effect similar to 2 but the new image slides in from the bottom and pushed the old images out. So, both the new and old images are sliding up. For implementing these two logics, a counter called `slideInBars` was used starting at value of 10. This variable was a counter to draw from 10 to 0 bars of 64px wide or high of the new upcoming image to draw the transition.

For the seventh requirement, the data was stored in a JSON array called `images`. The code is implemented in a generic way, so even if one were to add more images to the JSON array, the code would work.

For the eight requirements, please refer to the note at the very top of part 3.

For the ninth requirement, there are 20 images and their captions stored in JSON array called `images`.

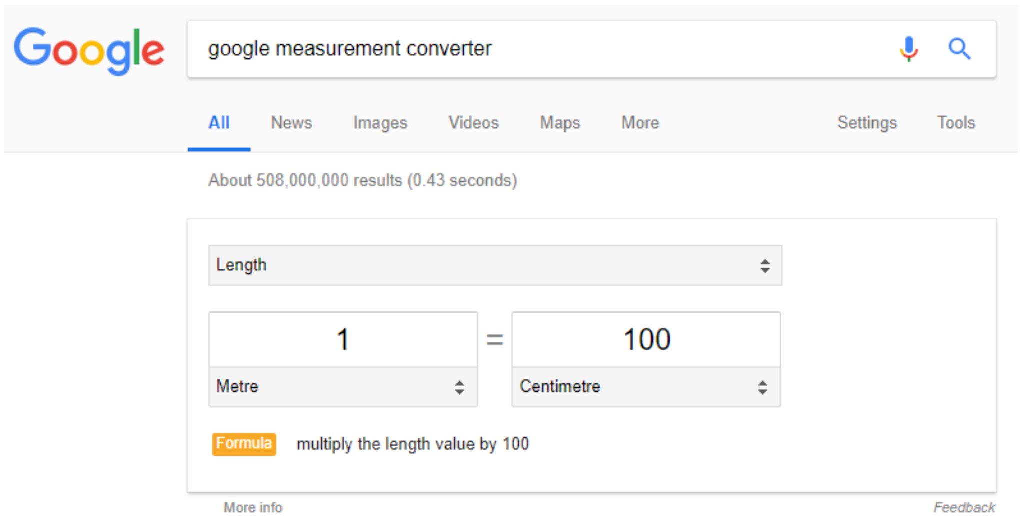
Part 4:

Hyperlink: `http://localhost/TMA_1/part4/part4.htm`

Three utility tools were created for this portion of the assignment. Measurement converter, mortgage calculator, and a Youtubeonrepeat File Generator. The six requirements were carefully implemented and exercised.

Measurement Converter

This tool was required to convert the following physical dimensions: Weights, Lengths, Areas, and Volumes. Please refer to the conversion tables containing the coefficients used to convert for these quantities. Please note, every combination was manually tested against Google's online Unit Converter tool.



About 508,000,000 results (0.43 seconds)

Length

1 = 100

Metre — Centimetre

Formula multiply the length value by 100

Weights Conversion Table

Before	After	Divide or Multiply	Factor
Kilogram	Gram	Multiply	1000
Kilogram	Pound	Multiply	2.205
Kilogram	Ounce	Multiply	35.274
Gram	Kilogram	Divide	1000
Gram	Pound	Divide	453.592
Gram	Ounce	Divide	28.35
Pound	Kilogram	Divide	2.205
Pound	Gram	Multiply	453.592
Pound	Ounce	Multiply	16
Ounce	Kilogram	Divide	35.274
Ounce	Gram	Multiply	28.35
Ounce	Pound	Divide	16

Lengths Conversion Table

Before	After	Divide or Multiply	Factor
Metre	Centimetre	Multiply	100
Metre	Foot	Multiply	3.281
Metre	Inch	Multiply	39.37
Centimetre	Metre	Divide	100
Centimetre	Foot	Divide	30.48
Centimetre	Inch	Divide	2.54
Foot	Metre	Divide	3.281
Foot	Centimetre	Multiply	30.48
Foot	Inch	Multiply	12
Inch	Metre	Divide	39.37
Inch	Centimetre	Multiply	2.54
Inch	Foot	Divide	12

Area Conversion Table

Before	After	Divide or Multiply	Factor
Square Metre	Acre	Divide	4046.856
Square Metre	Square Mile	Divide	2.59e+6
Square Metre	Square Foot	Multiply	10.764
Acre	Square Metre	Multiply	4046.856
Acre	Square Mile	Divide	640
Acre	Square Foot	Multiply	43560
Square Mile	Square Metre	Multiply	2.59e+6
Square Mile	Acre	Multiply	640
Square Mile	Square Foot	Multiply	2.788e+7
Square Foot	Square Metre	Divide	10.764
Square Foot	Acre	Divide	43560
Square Foot	Square Mile	Divide	2.788e+7

Volume Conversion Table

Before	After	Divide or Multiply	Factor
Litre	Cubic Metre	Divide	1000
Litre	US Liquid Gallon	Divide	3.785
Litre	US Tablespoon	Multiply	67.628
Cubic Metre	Litre	Multiply	1000
Cubic Metre	US Liquid Gallon	Multiply	264.172
Cubic Metre	US Tablespoon	Multiply	67628.045
US Liquid Gallon	Litre	Multiply	3.785
US Liquid Gallon	Cubic Metre	Divide	264.172
US Liquid Gallon	US Tablespoon	Multiply	256
US Tablespoon	Litre	Divide	67.628
US Tablespoon	Cubic Metre	Divide	67628.045
US Tablespoon	US Liquid Gallon	Divide	256

Mortgage Calculator

The mortgage calculator computed the periodic pay and the total interest fee for a given mortgage from the given principal amount in dollars, interest rate in percent value, amortization contract duration (in years), and the payment frequency. The formula below was used to compute for the periodic pay and total interest fees. Note the source used to implement this tool was from https://homeguides.sfgate.com/manually-calculate-house-payments-9633.html.

$$Periodic\ Pay = P * [\frac{r*(1+\mathrm{r})^2}{(1+\mathrm{r})^n - 1}],$$

$$Total\ Interest\ Fee = Periodic\ Pay * n - P,$$

Where P is the principal amount in dollars, r is the adjusted or normalized interest rate with respect to the periodic pay rate (periodic interest rate), and n is the number of payments. To normalize or adjust for r, simply divide by the total number of payments to occur in one year given the frequency monthly (12), weekly (52), or biweekly (26).

The above equation, and logic along with the appropriate DOM logic was used to implement for this tool.

Youtubeonrepeat File Generator

I have decided to implement a tool for generating Youtubeonrepeat files which has an extension of ".youtubeonrepeat". This file contains the user information containing the playlists the user has saved during their time in www.youtubeonrepeat.com, which I am a frequent user of. There is issue sometimes, when they would update their website, and sometimes the search bar in that site stops working when they spend a week updating something in their websites. This effectively renders users completely incapable of modifying or adding new playlist to their session information (stored in .youtubeonrepeat). Also, their website is designed so that you cannot save the same video in the same playlist more than once. This file format, after studying it was simply a JSON object, and the field JSON_data.userdata.playlists contained the list of all the playlists the user has saved as a list of JSON arrays. These JSON arrays contained the youtube videoID (the obfuscated code at the end of a YouTube link) and the title of the video.

The way I designed my tool was to use the Web API's FileReader() object to read local client's .youtubeonrepeat file. Once the client uploads the file, the user can click "Add new video via DOM" to add input fields for the new playlist's name, the youtube video titles, and the links. Once they fill the input fields in, they simply need to click the "Create File" button. The text field will be populated with the new JSON contents to be stored in the .youtubeonrepeat file. Once the user saves this data in any .youtubeonrepeat file of their liking, the just need to go to www.youtubeonrepeat.com, and upload the file containing this JSON data. Now the new playlist should be available for use. Make sure to add your favourite video multiple times since their website does not allow same video multiple times in the same playlist!

Also, if you need a sample Youtubeonrepeat File to test the app with, click on the "Sample Youtubeonrepeat File" tab to create a sample file with that link's JSON data. It contains couple playlists with cat and dog videos.